Sara Pinheiro de Brito

Integração de IPs em SoC e interface com memória

Campina Grande, Paraíba

Dezembro de 2018

Sara Pinheiro de Brito

Integração de IPs em SoC e interface com memória

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Universidade Federal de Campina Grande – UFCG

Centro de Engenharia Elétrica e Informática

Unidade Acadêmica de Engenharia Elétrica

Orientador: Prof. Dr. Marcos Ricardo Alcântara Morais

Campina Grande, Paraíba

Dezembro de 2018

Sara Pinheiro de Brito

Integração de IPs em SoC e interface com memória

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Campina Grande, Paraíba
Dezembro de 2018

Agradecimentos

Agradeço, primeiramente, aos meus pais, que sempre se esforçaram para que eu pudesse ter as melhores oportunidades de ensino. Agradeço também ao meu irmão, por estar presente sempre que precisei.

Agradeço a Lucas, que me acompanhou em praticamente toda a jornada que foi esse (e está sendo) esse curso. Muito obrigada pela (enorme) paciência de sempre.

Agradeço aos professores orientadores (prof. Marcos e prof. Gutemberg) do Laboratório em Excêlencia em Microeletrônica no Nordeste (XMEN) por fazerem a diferença em nosso meio acadêmico e buscarem sempre mais - o que, por consquência, fez o laboratório nascer. Agradeço também pela oportunidade de trabalhar no laboratório, sem dúvida, foi a oportunidade mais ímpar que pude ter durante a graduação.

Agradeço aos todos membros do XMEN, com os quais aprendi muito. Microeletrônica é muito lindo, mas com certeza a experiência dentro do laboratório foi mais legal pelo ambiente que criamos, que só foi possível porque todo mundo era meio doido.

"Any form of human creativity is a process
of doing it and getting better at it.
You become a writer by writing.
There is no other way. So do it, do it more.
Do it better. Fail. Fail better."
(Margaret Atwood)

Resumo

O princípio da Internet das Coisas (*Internet of Things* – IoT) é que essa nova tecnologia irá conectar bilhões de dispositivos usando a internet começando por volta de 2020. O sistema IoT consiste em três componentes principais: as coisas ou recursos (sensores), as redes de comunicação que as conectam (conectividade) e os sistemas de computação, que usam os dados que fluem de e para as nossas coisas (aplicações em software).

Com o intuito de acompanhar esta ascensão do IoT, foi criado o Projeto para Excelência em Microeletrônica (PEM) com objetivo de desenvolver um *System on Chip* composto por um microprocessador e alguns IPs. Este trabalho tem como objeto realizar a integração destes IPs neste *System on Chip* e o desenvolvimento do seu mecanismo de inicialização.

Palavras-chave: Microeletrônica. *System on Chip*. *Internet of Things*.

Abstract

The principle of Internet of Things (IoT) is that this new technology will connect billions of devices by 2020. The IoT system is composed by three main components: the ”things“ or resources (the sensors), the communication networks that interconnect them (connectivity) and the computing systems, which uses the data that flow to and from these things (software applications).

As a form of accompany the rise of IoT, the Projeto para Excelência em Microeletrônica (PEM) was conceived, with the main goal of developing a System on Chip composed by a microprocessor and some IPs. This work focuses on peforming these IPs integration on that System on Chip and on developing its boot mechanism.

Keywords: Microelectronics. System on Chip. Internet of Things.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Breve História da Microeletrônica

No último século foi possível acompanhar o crescimento da tecnologia, foi possível observar a tecnologia entrar no dia a dia das pessoas. Por exemplo, hoje é difícil de imaginar um mundo sem computador, celular ou internet. E todo esse avanço se tornou possível por causa de um dispositivo: o transistor.

O primeiro transistor foi criado em 1947, por John Bardeen e Walter Brattain no *Bell Laboratories* (Figura 1a). Dez anos depois, Jack Kilby fez o primeiro circuito integrado, na *Texas Instruments*, composto por alguns transistores em uma única peça de silício (Figura 1b). Assim como a invenção do transistor, a invenção do circuito integrado garantiu o prêmio Nobel de Física para seus inventores.

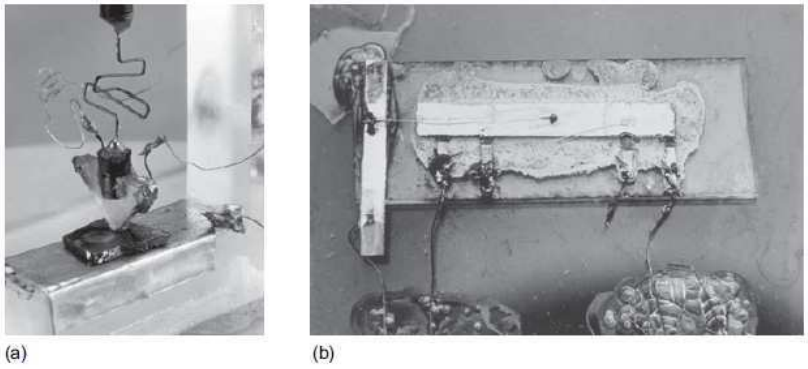

Figura 1: (a) Primeiro transistor (Arquivos da AT&T) e (b) primeiro circuito integrado (Arquivos da *Texas Instruments*). Fonte: Weste e Harris (2011)

Nos anos seguintes, tanto o transistor continuou diminuindo quanto o processo de fabricação dos circuitos integrados continuou sendo melhorado. Em 1965, Gordon Moore observou que o número de transistores em uma mesma área estava dobrando a cada 18 meses. Isso acabou sendo conhecida como a Lei de Moore e virou uma profecia "auto-realizável".

Aliado a melhoria na fabricação, milhares de transistores integrados em um circuito integrado são capaz de armazenar informações ou executar centenas (ou milhares) de operações por segundo. Por isso, o transistor e, por consequência, os circuitos integrados podem ser considerados uma das maiores histórias de sucesso do século.

O nível de integração dos circuitos integrados podem ser classificados como: escala pequena (*small scale*), escala média (*medium scale*), escala alta (*large scale*) e escala muito alta (*very large scale*). Os circuitos em pequena escala de integração (*Small Scale*

Integration - SSI) contavam com até 10 portas lógicas, sendo em torno de seis transistores por porta. Já a escala média de integração (MSI), contava com até 1000 portas lógicas e a escala alta (LSI) com até 10000 portas. Ficou notável que novos nomes teriam que ser criados a cada cinco anos se esta tendência de nomenclatura continuasse e assim o termo integração em larga escala (VLSI) é usado para descrever a maioria dos circuitos integrados a partir dos anos 80 em diante. Na Figura 2 é possível ver um exemplo do crescimento rápido dessa tecnologia anualmente.

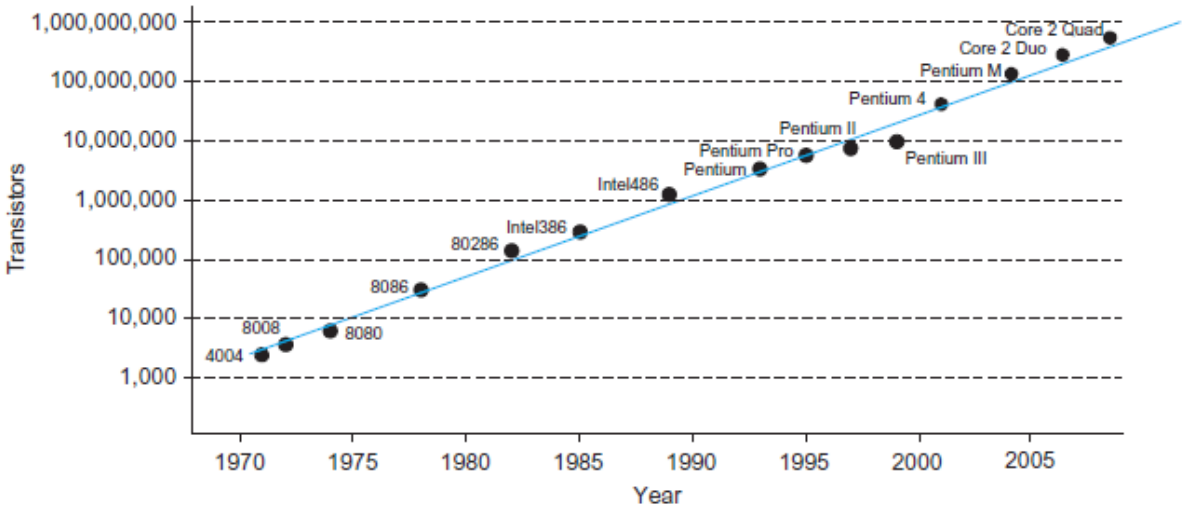

Figura 2: Transistores nos microprocessadores da Intel. Fonte: Weste e Harris (2011)

1.2 Sistemas Integrados em Chip

> Sistema embarcado - É uma combinação entre *hardware* e *software* de um computador e, as vezes, pode incluir mecânicas adicionais ou outras peças, projetadas para executar uma função dedicada. Muitas vezes, os sistemas embarcados fazem parte de um sistema ou produto maior, como no caso de um sistema de freios antitravamento em um carro. (STALLINGS, 2013)

Um *System on Chip* (SoC) é um sistema embarcado no qual os componentes de um computador ou outros sistemas eletrônicos foram integrados em um *chip*. Similar ao computador, SoCs incluem uma CPU, memória e componentes de entrada e saída. Esses componentes são interconectados para realizar a função básica de executar um programa.

Uma Unidade Central de Processamento (*Central Processing Unit* - CPU) pode ser vista como um circuito de propósito geral que realiza funções lógicas e aritméticas. O resultado produzido por este circuito depende dos sinais de controle aplicados a ele (Figura 3). Por esse motivo, é possível escrever conjuntos diferentes de instruções para

que o *hardware* interprete e gere sinais de controle. Esse conjunto de instruções é chamado de *software*.

Ao longo dos anos, escrever essas instruções foi ganhando níveis maiores de abstração, tornando fácil para o desenvolvedor que escreve o *software* abstrair o circuito por trás dos processadores. Essa facilidade de programar e re-programar o mesmo circuito aumenta a escalabilidade do *chip* enquanto produto e é um dos grandes motivos para a eletrônica digital ser um sucesso.

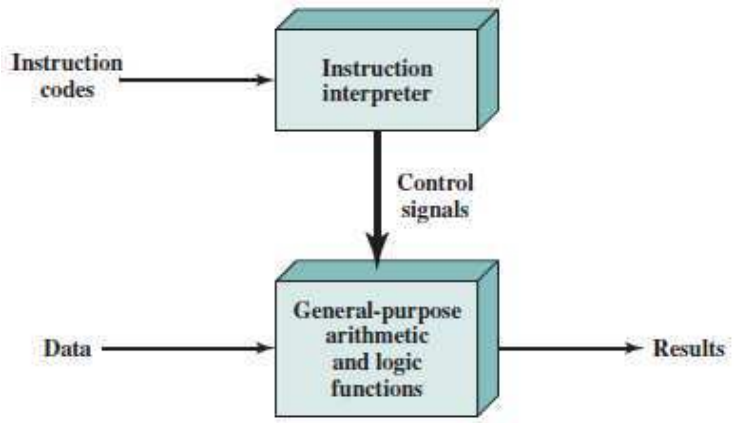

Figura 3: Programar em software. Fonte: (STALLINGS, 2013)

Além da facilidade de programar, a indústria de microeletrônica também cresceu com a facilidade de replicar *hardwares*. Pode-se ver o *hardware* como um bloco lógico fechado e combiná-lo com algum protocolo de comunicação torna-o compatível a qualquer outro sistema que entenda esse protocolo.

Normalmente, os *hardwares* escaláveis são *chips* que realizam uma determinada tarefa, definidos na microeletrônica como propriedade intelectual (*Intellectual Property* - IP). Assim, é possível reutilizar o mesmo bloco em projetos diferentes, aumentando sua escalabilidade. Por exemplo, um IP de criptografia pode estar no sistema de uma biblioteca, de um carro, de um celular, bastando apenas que o IP consiga se comunicar com a unidade de controle do sistema.

1.3 Projeto PEM

O Projeto para Excelência em Microeletrônica (PEM) surgiu na Universidade Federal de Campina Grande (UFCG) para trabalhar nessa área tão importante para tecnologia nas últimas décadas. Juntamente ao projeto, criou-se o Laboratório de Excelecência em Microeletrônica no Nordeste (XMEN).

Este primeiro projeto teve como objetivo capacitar os alunos para produzir um *System on Chip* para aplicação em *Domestic Internet of Things* (DIoT) e é composto pelos

seguintes IPis (*Intellectual Property*):

- Núcleo RISC-V RI5CY;
- SPI;
- I2C;
- UART;
- Timer;
- Criptografia (AES);
- Correção de erros (Viterbi e Reed-Solomon);
- *Power Line Comunication* (PLC)

Os três últimos IPs foram desenvolvidos pelo Laboratório XMEN e Laboratório de Arquiteturas Dedicadas (LAD). O núcleo RISC-V e os outros periféricos foram desenvolvidos pela universidade ETH Zurich (https://www.pulp-platform.org), no projeto nomeado de PULP (*Parallel Ultra Low Power*).

O objetivo do PULP é desenvolver uma plataforma de pesquisa e desenvolvimento com *hardware* e *software* abertos, escaláveis e com baixo cosumo de potência, da ordem de miliwatts. Além disso, tem como objetivo satisfazes as demandas computacionais de aplicativos de *Internet of Things* (IoT).

Quando o PEM começou, o PULP só havia lançado um *single-core*, que é o RI5CY, então, foi com ele que trabalhamos. Além disso, a escolha para trabalhar com o *core* RI5CY deu-se exatamente pelo objetivo do projeto PULP: primeiro, é um *hardware* aberto; segundo, tem como foco baixa potência e atender a demandas de aplicações IoT.

O objetivo deste trabalho é fazer a integração dos IPs desenvolvidos pela UFCG com o processador RISC-V RI5CY do projeto PULP.

2 Embasamento Teórico

2.1 Inicialização

Executar um programa é seguir uma sequência de etapas. Em cada etapa, alguma operação aritmética ou lógica é executada em alguns dados e para cada etapa é necessário um novo conjunto de sinais de controle. Essa sequência de etapas é descrita, para o *hardware* através de uma sequência de instruções. Essas instruções ficam guardadas na memória do computador.

Quando fala-se de memórias, há dois tipos de arquitetura de computadores: Arquitetura de von Neumann e Arquitetura de Havard. A primeira é composta por uma memória para dados e para instruções, já a segunda consiste em duas memórias separadas, uma para instruções e outra para dados. No caso desse projeto, a arquitetura utilizada é a que foi adotada pelo projeto do PULPino, a Harvard.

Ao ser ligado, o *hardware* busca a primeira instrução em um endereço fixo da memória de instruções para executar o primeiro programa, que é conhecido como "programa carregador de partida" (*BootStrap Loader*). Para que isto aconteça, é necessário que seja alocado um espaço na memória exclusivamente para este programa, o *BootStrap Loader*. Este processo de inicialização de um computador, ou de um SoC, é conhecido pelo termo em inglês *boot* e o espaço alocado na memória para guardar as instruções do programa é conhecido como memória de *boot*.

Em máquinas de uso geral, o programa de partida é usado para iniciar e carregar os serviços de interface de entrada e saída, que são fornecidos em um nível físico (menos abstrato), e para carregar o Sistema Operacional (*Operating System - OS*), que também fornece serviços de interface de entrada e saída em um nível lógico (mais abstrato). A partir desse ponto, o OS cria um "Ambiente de Execução" para que o usuário possa interagir mais amigavelmente com a máquina.

Já em máquinas de uso específico, o programa de partida pode carregar um programa de controle que controla de forma autônoma toda a operação dos sistema (controlando, inclusive, todas as interfaces de entrada e saída). Esse programa de controle pode residir completamente em memória ROM e também pode carregar um "Sistema Operacional", cujos serviços são usados pelo "programa de controle".

2.2 Barramentos

Para que o SoC ou um computador funcione é necessário que haja comunicação entre seus componentes. Por exemplo, a CPU tem que mandar e receber sinais de um teclado para que ele cumpra a tarefa de escrever algo na em uma tela. Outro exemplo, é a comunicação com as memórias de instruções e de dados para que qualquer programa seja executado, é necessário que haja comunicação entre memórias (de instruções e às vezes de dados) e CPU.

Essa comunicação entre a CPU e a memória ou os periféricos acontece no **barramento**, um conjunto de linhas compartilhadas para troca de palavras digitais. Na Figura 4, é exemplificado como é essa arquitetura tradicional de barramento: um único barramento compartilhado para CPU, memória e entradas e saídas.

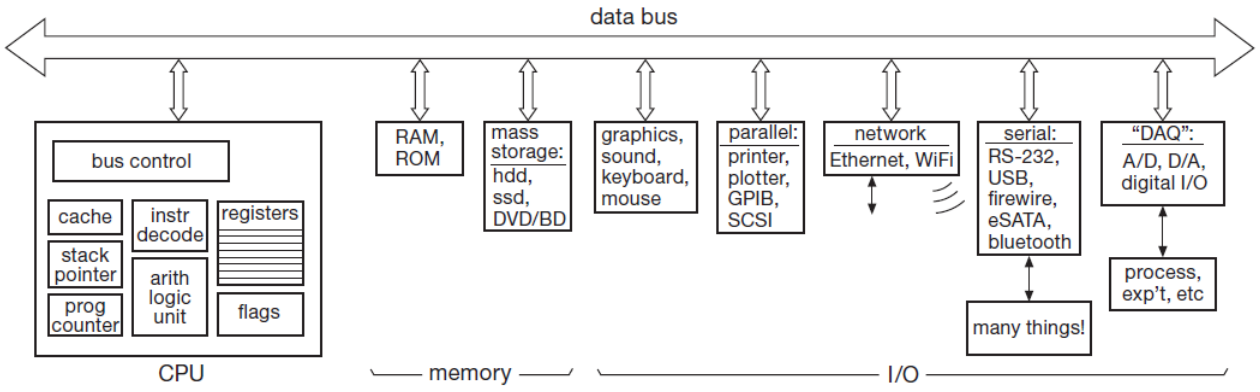

Figura 4: Arquitetura tradicional de um barramento. Fonte: (HOROWITZ; HILL, 2016)

Normalmente, um barramento é composto por linhas de dado, de endereço e de controle que são resposáveis por transmitir os dados seguindo determinadas regras de acordo com o protocolo de comunicação adotado. Além disso, quando fala-se de barramento, frequentemente o emissor é denominado de *master* e o receptor de *slave*, ou, em português, mestre e escravo.

Tradicionalmente existe nas linhas de controle um clock e um protocolo fixo de comunicação que apresenta sincronia com clock. Um exemplo simples seria um protocolo que consiste em um mestre enviar um comando de leitura e um endereço em um clock e receber do escravo o resultado dessa leitura em um clock posterior. A única limitação dessa lógica é que cada dispositivo precisa operar na mesma frequência do clock e, quanto mais dispositivos houver, maior será o caminho que o clock irá percorrer e mais difícil será atingir o sincronismo físico dos sinais.

Também é possível que o protocolo de comunicação seja assíncrono. Para controlar a transmissão de dados nesses tipos de protocolo, é necessário ter um mecanismo conhecido como *handshake*, do inglês, "aperto de mão". Esse mecanismo consiste no emissor e

receptor concordarem que a transação ocorreu, por meio de envio de sinais elétricos pré-estabelecidos.

Também é possível que um mesmo protocolo utilize do síncronismo do clock e do processo de *handshake* para se tornar um protocolo consistente. Por exemplo, não necessariamente um mestre e um escravo precisariam operar na mesma frequência de clock se houver um mecanismo de *handshake* associado ao procolo síncrono.

O protocolo de comunicação utilizado neste trabalho foi o AXI4-Lite (Advanced Extensible Interface 4 Lite), que é um dos protocolos AMBA (Advanced Microcontroller Bus Architecture) da ARM. Os protocolos AMBA são especificações de interconexão para *chips*, criados pela ARM e, além de serem bem documentados, também são livres, podendo serem utilizados sem o pagamento de *royalties*.

O protocolo AMBA AXI suporta *designs* de sistemas de alta performance e alta frequência, além de fornecer flexibilidade na implementação de arquiteturas de interconexão e é compatível com versões anteriores das interfaces AHB e APB existentes.

Os principais recursos do protocolo AXI são:

- suporte para transferências de dados não alinhadas, usando strobes de bytes;
- separar os canais de leitura e escrita de dados, que podem fornecer acesso direto à memória (*Direct Memory Access* DMA);
- suporte para a emissão de vários endereços pendentes;
- suporte para conclusão de transação fora de ordem;
- permite fácil adição de etapas de registro para fornecer fechamento de tempo.

O protocolo AXI é definido com cinco canais independentes: endereço de leitura, leitura de dados, endereço de escrita, escrita de dados e de resposta de escrita.

Para uma transação de leitura, é necessária a atuação de dois canais: o de endereço de leitura e o de leitura de dados. O mestre envia pelo canal de endereço de leitura o endereço que se quer ler e o escravo usa o canal de leitura de dados para retornar o dado solicitado. É possível ver um diagrama do fluxo de leitura na Figura 5.

Já para uma transação de escrita, é necessária a atuação de três canais: endereço de escrita, escrita de dados e de resposta de escrita. Os canais de dados e endereço transferem, do mestre para o escravo, o dado e o endereço no qual este será escrito, respectivamente. Para finalizar a transação, o escravo usa o canal de resposta de escrita para sinalizar a conclusão da transferência para o mestre. Na Figura 6 vê-se um diagrama da operação de escrita.

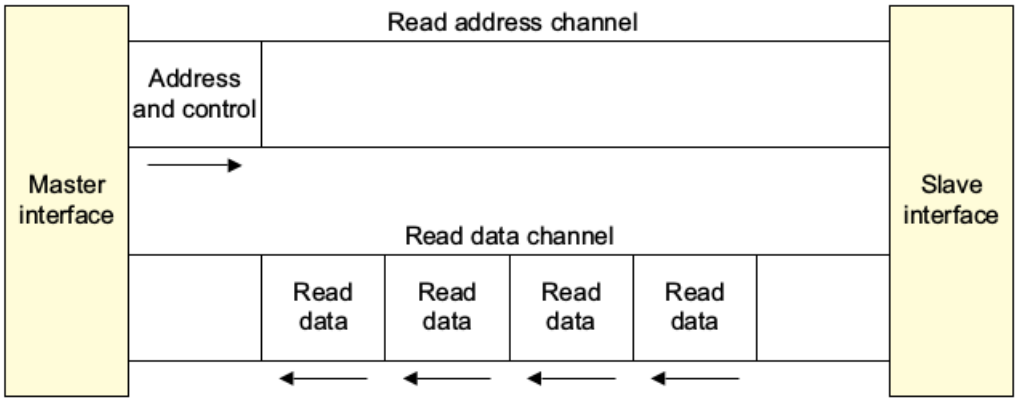

Figura 5: Canal de leitura do protocolo AXI. Fonte: (ARM, 2011)

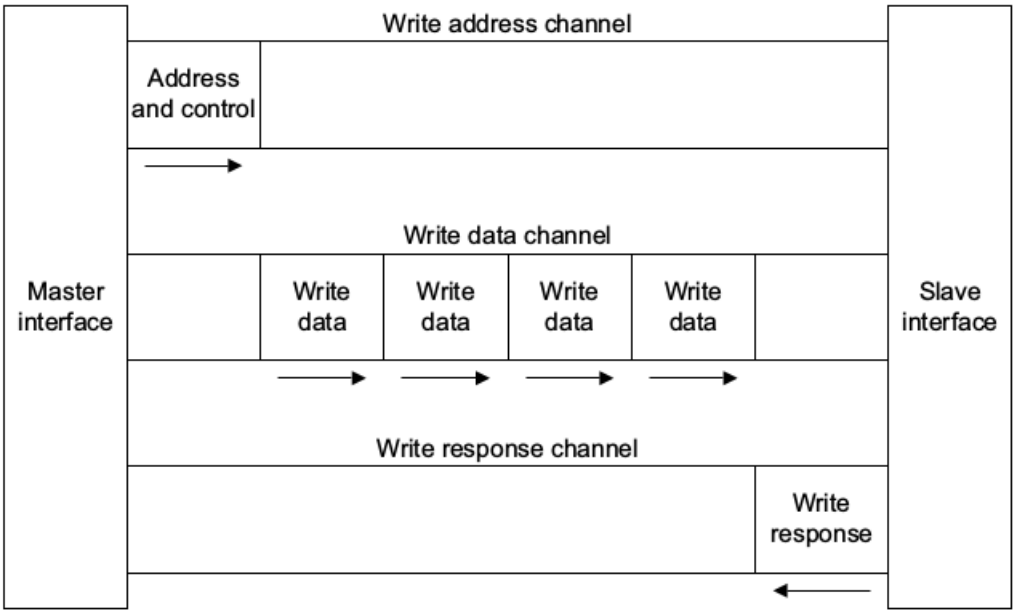

Figura 6: Canal de escrita do protocolo AXI. Fonte: (ARM, 2011)

2.3 Pulpino

O PULPino é um *System on Chip* com *single-core* desenvolvido pela *Integrated Systems Laboratory* (IIS) de *ETH Zürich* e *Energy-efficient Embedded Systems* (EEES) da *University of Bologna*. Na Figura 7 pode-se ver o diagrama de blocos do PULPino. O *core* se comunica diretamente com as memórias (de instruções e de dados) e por meio de um barramento AXI4 com todo o resto do *chip*. Do ponto de vista do barramento AXI, tem-se três *masters*: o *core*, o IP *SPI Slave* e o IP *Advanced Debug Unit*.

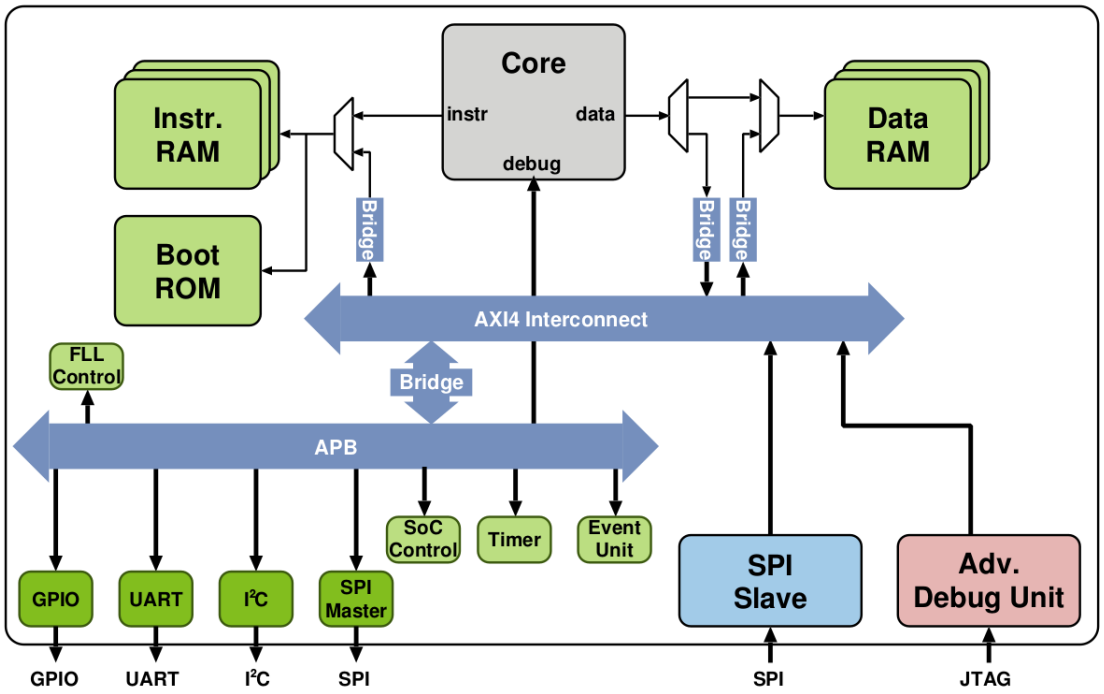

Figura 7: Diagrama de blocos do Pulpino. Fonte: (TRABER; GAUTSCHI, 2017)

O *core* também foi desenvolvido pelas mesmas universidades e usa um conjunto de instruções (*Instruction Set Architecture* - ISA) RISC-V. O RISC-V é uma ISA aberta desenvolvida originalmente na Divisão de Ciência da Computação da *University of California, Berkeley*. Ela, por sua vez, é baseada nos princípios de computação de conjunto de instruções reduzidas (RISC), assim como o ARM e o MIPS e outras arquiteturas comuns de processadores comerciais.

2.4 IPs do Laboratório XMEN

A seguir são expostas breves descrições dos IPs desenvolvidos pelo Laboratório XMEN para o SoC. Estas descrições foram retiradas do documento "Especificação SoC IoT - Chilipe", que pode ser encontradas no repositório *git* do projeto.

Os IPs AES, Reed Solomon e Viterbi foram desenvolvidos completamente no Laboratório XMEN. Já o PLC teve sua maior parte desenvolvida no LAD com o projeto Brazil IP, sendo apenas finalizado no Laboratório XMEN.

2.4.1 AES

O *Advanced Encryption Standard* (AES) é um algoritmo de criptografia de chave simétrica, desenvolvido pelo governo dos EUA e anunciado pelo NIST (Instituto Nacional de Padrões e Tecnologia dos EUA) como U.S. FIPS PUB (FIPS 197). O AES cifra/decifra

informações de 128 *bits*, com chaves de tamanhos variados, podendo ser de 128, 192 ou 256 *bits*.

O IP desenvolvido implementa esse algoritmo, oferecendo um hardware dedicado para transmitir e receber dados com segurança, porém só permite chaves com tamanhos de 128 *bits*. Esse IP contempla 6 modos de operação: ECB, CBC, PCBC, CFB, OFB e CTR. Esses modos realizam algumas operações simples, como por exemplo operação de XOR com o resultado da encriptação/decriptação com algum outro valor. O objetivo é proteger a informação de possíveis ataques *hackers*. Cada modo tem suas peculiaridades em relação ao tipo de proteção, ficando ao critério do programador escolher o mais adequado para sua aplicação.

2.4.2 Reed-Solomon

Os códigos de Reed-Solomon (RS) podem ser definidos como um conjunto de códigos redundantes de correção de erro com natureza linear e de bloco profundamente baseada na álgebra de campos finitos (ou campos de Galois). Tais características tornam o código particularmente poderoso na correção de erros em rajada tendo em vista que, sendo um código de correção em blocos, a mensagem a ser transmitida é dividida em pacotes individuais de informação (denominada símbolos), e a redundância, primordialmente aplicada em cada *bit*, passa a ser interpretada como um dos símbolos exclusivos que irá compor o bloco final da mensagem. Caso um símbolo esteja completamente errado, a correção ocorre no símbolo e contabilizará como apenas um erro, e não um para cada *bit* que compõe o símbolo.

Assim sendo, os métodos de correção de erro de tais algoritmos são especificados a partir da quantidade total de símbolos (`n`) e da quantidade de símbolos que representam a informação real a ser enviada (`k`) em um único bloco de informação. A Figura 8 exemplifica o formato final de um bloco de informações codificado, ou seja, com os *bits* de informação agrupados em diferentes símbolos de mesma dimensão e os *bits* de redundância agrupados em símbolos de dimensão semelhantes e adjacentes uns aos outros - dando a impressão de uma única informação de `n` *bits*, o que na verdade são informações de $n \times m$ bits agrupados em `n` símbolos. Configurações comuns de utilização do RS são o DVB-T (255,239) e o sistema de DVD (208,192).

O IP, para esse caso, implementa um codificador e um decodificador RS configurado com $m = 8$, $n = 31$, $k = 23$ e $2t = 8$. Consequentemente, o RS do IP trabalha com mensagens de 23 símbolos de 1 byte cada e uma paridade de 8 símbolos de 1 byte cada - transferindo um total de 118 *bits* por pacote de mensagem com uma capacidade de correção de 4 símbolos (`t`) completos. Em suma, implementa-se um RS (31, 23) com $2t = 8$. É importante salientar que, matematicamente, o sistema garante identificação e correção de erro para até 4 símbolos. Para mensagens que apresentem um número maior do que

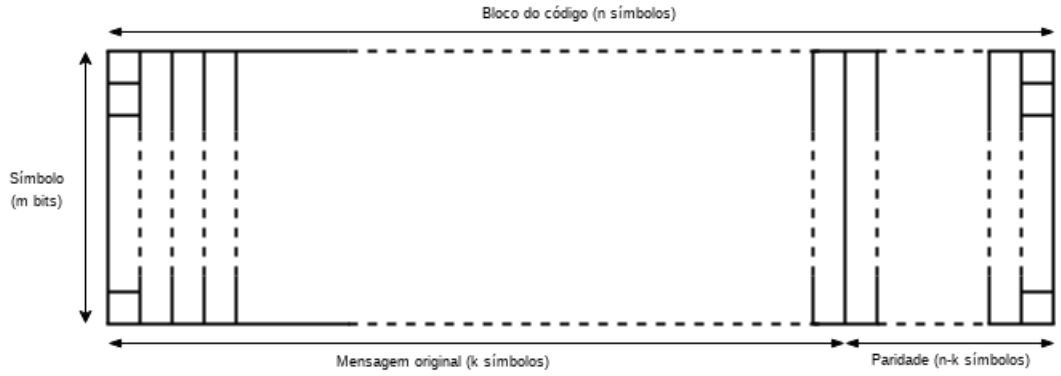

Figura 8: Exemplo de um bloco codificado de informação.

quatro símbolos com erro, os resultados apresentados não podem ser confiáveis - embora, em alguns casos, a mensagem decodificada esteja correta.

2.4.3 Viterbi

O Viterbi é um algoritmo dinâmico de programação para encontrar a sequência de estados mais provável -chamada de caminho Viterbi. Esse é utilizado de forma universal em decodificar códigos convolucionais usados em celulares digitais CDMA (Acesso Múltiplo por Divisão de Código) e GSM (Sistema Global para Comunicações Móveis), satélites, comunicação de probes no espaço e LANs sem fio 802.11.

O IP do Viterbi é responsável por executar operações de correção de erros de dados recebidos. Para isso, antes de enviar dados para o canal analógico, o código cria símbolos de paridade via aplicação de uma função booliana polinomial para um fluxo de dados. Este IP é capaz de operar em três modos distintos: Perfuração, duplo e triplo, no qual 2/3, 1/2 e 1/3 é a taxa de código de cada, respectivamente.

2.4.4 PLC

O PLC é um módulo que implementa a camada física da norma internacional IEC 61334-5-1. O princípio da comunicação via rede elétrica é baseada na transmissão de dados em esquema de modulação S-FSK de forma sincronizada com a frequência da rede.

Os principais submódulos deste IP são o modulador, que é composto por uma tabela contendo amostras de senoide e um conversor digital-analógico do tipo sigma-delta, e o demodulador, que implementa o algoritmo de Goertzel avaliar o valor do sinal nas frequências de operação definidas pelo usuário.

3 Arquitetura de Integração

A arquitetura proposta para o SoC é a que se encontra na Figura 9. Todos os IPs do projeto havia sido projetados utilizando o protocolo AXI-4 Lite e, por esse motivo, o primeiro passo para integração foi desenvolver um *interconnect* para conectar o *core* aos IPs.

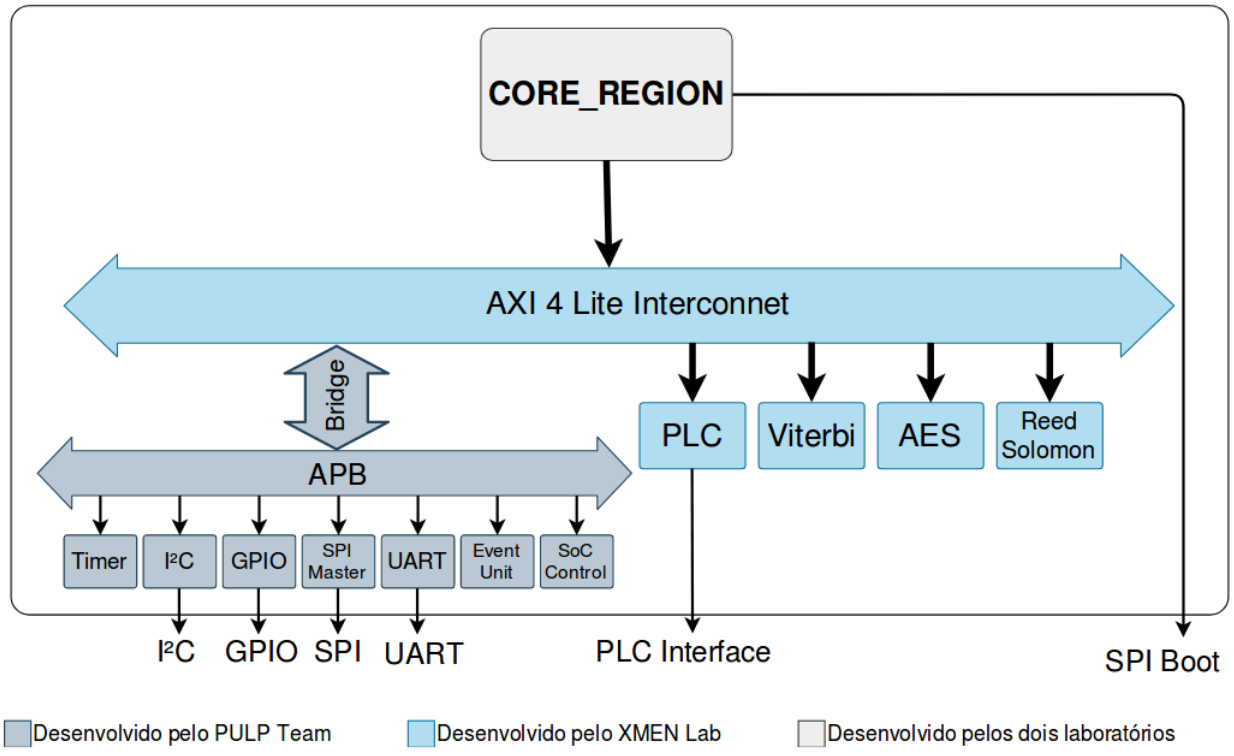

Figura 9: Diagrama de blocos do SoC Chilipe.

3.1 Protocolo AXI4 Lite

Nesse projeto, foi adotado o protocolo AXI4 Lite, que é um subconjunto do protocolo AXI que possui uma interface de controle mais simples.

Como já foi explicitado na Seção 2.2, a interface do AXI4 contém 5 canais e a interface do AXI4 Lite não é diferente. Os sinais de cada canal podem ser vistos na Figura 10. Basicamente, há os sinais de *handshake* para todos os canais e os típicos sinais de dado e endereço. Além desses sinais, há mais três tipos de sinais:

AWPROT ou ARPROT Tipo de proteção. Este sinal indica o nível de privilégio e segurança da transação e se a transação é um acesso de dados ou um acesso de

instrução.

WSTRB *Strobe* de escrita. Este sinal indica quais faixas de *bytes* contêm dados válidos. Há um bit de *strobe* para cada oito bits do barramento de dados a ser escrito.

BRESP ou RRESP Resposta de escrita/Resposta de leitura. Este sinal indica o status da transação de escrita/leitura.

Além desses canais, há os sinais globais de *clock* (CLK) e de *reset* (ARESETn), este último é baixo ativo. A interface deve obeceder as seguintes regras sobre *clock* e *reset*:

- Todas as entradas amostradas na borda de subida do *clock*;
- Todas as saídas são mudadas após a borda de subida do *clock*;
- Durante o *reset*, o mestre deve aplicar nível lógico baixo para os sinais de ARVALID, AWVALID e WVALID. Já o escravo deve aplicar nível lógico baixo para os sinais RVALID e BVALID.

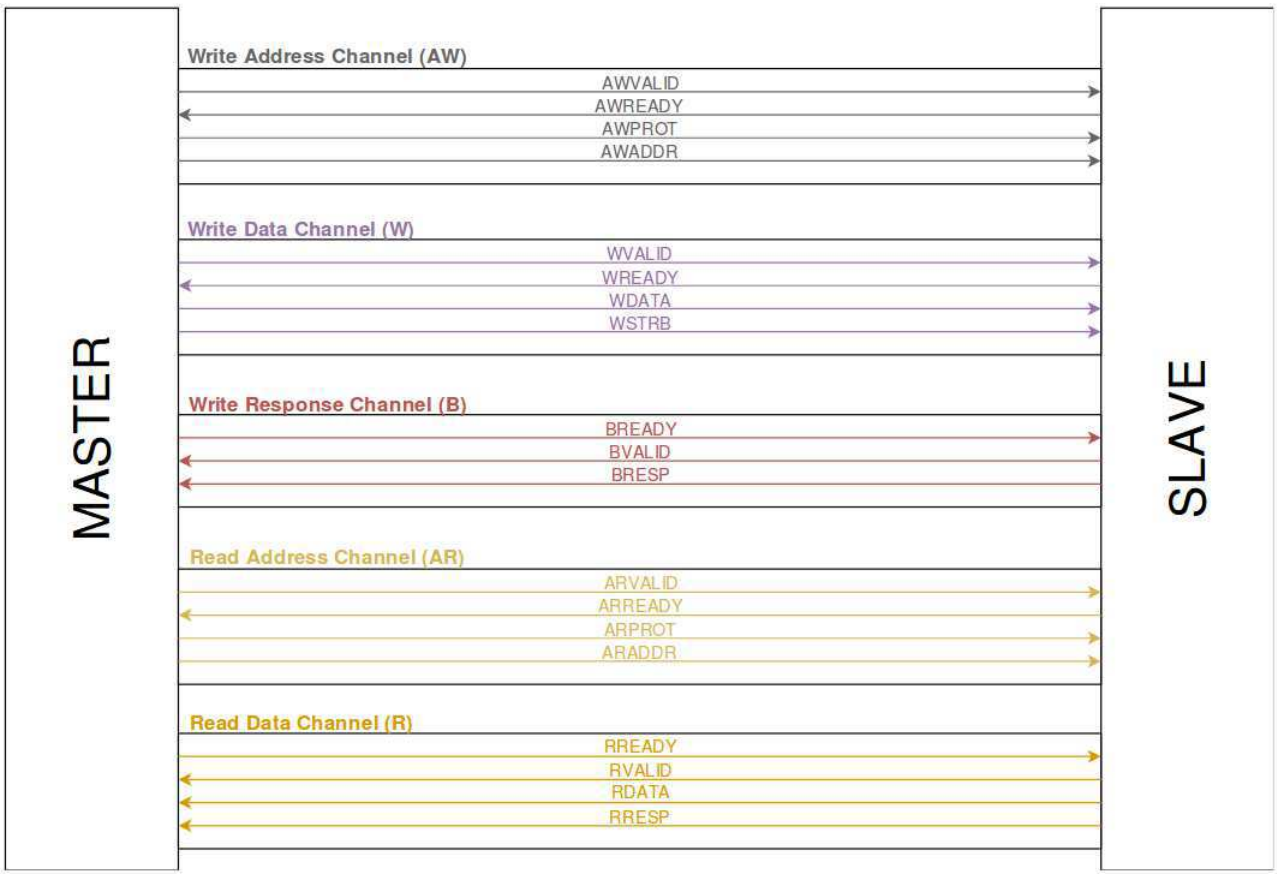

Figura 10: Interface do AMBA AXI4 Lite.

3.1.1 Processo de *handshake*

Todos os cinco canais de transação usam o mesmo processo de *handshake* VALID/READY para transferir informações de endereço, dados e controle. Esse mecanismo de controle de fluxo bidirecional significa que tanto o mestre quanto o escravo podem controlar a taxa na qual a informação se move entre o mestre e o escravo. A fonte gera o sinal VALID para indicar quando o endereço, dados ou informações de controle estão disponíveis. O destino gera o sinal READY para indicar que pode aceitar as informações. A transferência ocorre apenas quando os sinais VALID e READY estão em nível lógico alto.

Além do processo de *handshake*, também é necessário seguir as seguintes regras do protocolo:

1. Não é permitido a uma fonte aguardar que o "READY" seja declarado antes de declarar "VALID" (Figura 11);
2. Uma vez que o "VALID" é declarado, deve permanecer assim até que o *handshake* aconteça em uma borda de subida do *clock* (Figura 11);
3. É permitido que o destino espere que "VALID" seja declarado antes de declarar o "READY" correspondente;
4. Se "READY" for declarado, é permitido desabilitá-lo antes que "VALID" seja declarado.

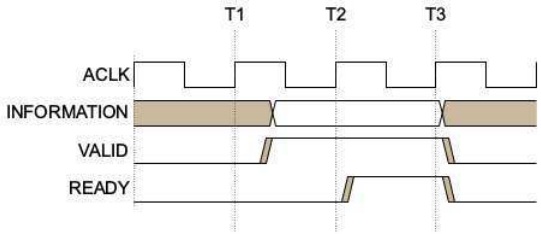

Figura 11: VALID antes de READY.

3.1.2 *Interconnect* - o funcionamento

O *interconnect* desenvolvido suporta um mestre e vários escravos. Para o SoC, o mestre é o *core* e o *interconnect* funciona como um grande "demultiplexador", que decodifica com qual escravo o mestre está tentando se comunicar e faz as transições corretamente.

A base do funcionamento do *interconnect* está descrita na Figura 12. Apesar do diagrama descrever o caso apenas para os canais de leitura, a lógica para os canais de

escrita é exatamente a mesma. Como o protocolo de comunicação deve ser seguido pelo mestre e pelo o escravo, o *interconnect* pode abstrair o processo como um todo e se importar apenas com o começo e final das transações.

No caso, tanto para leitura quanto para escrita, a transação começa com o mestre enviando o endereço da transação a ser feita. Então, enquanto não há endereço válido, a máquina de estados do *interconnect* fica no estado WAIT_ADDR. No caso de endereço válido, o *interconnect* fica no estado WAIT_RESP esperando que ocorra o *handshake* dos canais de resposta.

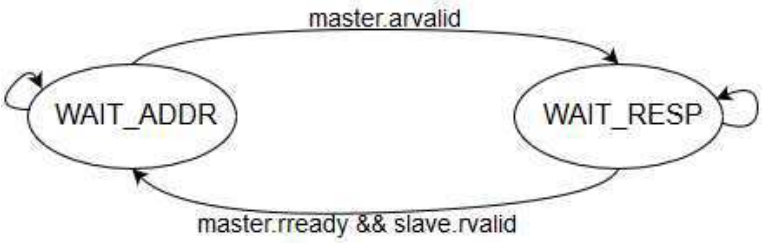

Figura 12: Máquina de estados do *interconnect*.

Como após o *handshake* do canal de endereço, o mestre pode já começar a solicitar outra transação em outro endereço, foi adicionado um *buffer* para salvar o endereço da transação em andamento para que ela termine corretamente.

3.1.3 *Interconnect* - o mapa de memórias

Outra parte do *interconnect* corresponde ao mapeamento de endereços. Parte desse mapeamento foi herdado do projeto do PULPino, sendo acrescentado os endereços para os IPs desenvolvidos pelo Laboratório XMEN. O mapa de memória do sistema está apresentado nas Tabelas 1 e 2.

O endereçamento para os IPs, como pode-se perceber pelas tabelas, dá-se pelos *bits* mais significativos do endereço. Para os espaços na memória não-mapeados foi criado um IP escravo cuja função é responder que houve erro de decodificação, seguindo o protocolo AXI4 Lite.

0x0000_0000	Memória de instrução
0x0000_8000	SPI Boot
0x0000_FFFF	
0x0010_0000	32kB Data RAM
0x0010_8000	
0x1910_0000	Reed-Solomon
0x1910_1000	Viterbi
0x1910_2000	AES
0x1910_3000	SPI IP
0x1910_4000	PLC
0x1A10_0000	Peripherals*

Tabela 1: Mapa de memória

0x1A10_0000	UART
0x1A10_1000	GPIO
0x1A10_2000	SPI Master
0x1A10_3000	Timer
0x1A10_4000	Event/Interrupt Unit
0x1A10_5000	I2C
0x1A10_6000	Não Usado
0x1A10_7000	SoC Control

Tabela 2: Mapa de memória - Peripherals

3.2 Integração para sistema de *boot*

Relembrando a Figura 7, o PULPino é constituído por um *interconnect* que permite mais de um mestre no barramento e a memória de *boot* ficava acessível para escrita através do barramento AXI, e pode ser escrita pelo "SPI *slave*".

No caso deste projeto, o *interconnect* tem a limitação de suportar apenas um mestre, impedindo de ser reutilizado o mesmo sistema de *boot* do PULPino. Foi cogitado voltar a estrutura original do PULPino, com o *interconnect* AXI 4 *Full*. No entanto, os IPs feitos pelo projeto são AXI4-Lite e para compatibilizar as duas interfaces, seria necessário traduzir ou retornar erro nas operações mais complexas de AXI (Tabela 3). As duas opções foram descartadas, pois eram iniviáveis dado o prazo existente para finalização do projeto.

A primeira opção de retornar erro já exigiria um esforço de verificação para garantir que o resultado é compatível com o padrão AXI. Mesmo sendo compatível, seria preciso de avaliar o impacto deste comportamento no sistema. Por exemplo, o *core* ignora a resposta das transações AXI e, em caso de erro, ele apenas lê um valor inválido ou pensa que escreveu. Há o risco de a interação do *core* com o *interconnect* gerar falhas imprevisíveis de escrita/leitura.

Já a opção de traduzir as operações evitaria esse último problema, e até tornaria os IPs acessíveis pelo JTAG, mas envolveria um esforço ainda maior não só de verificação como de projeto.

Mestre	**Escravo**	**Interoperabilidade**
AXI	AXI	Totalmente operacional.
AXI	AXI4-Lite	É necessário refletir o AXI ID. Conversão pode ser necessária.
AXI4-Lite	AXI	Totalmente operacional.
AXI4-Lite	AXI4-Lite	Totalmente operacional.

Tabela 3: AXI e AXI4-Lite Interoperabilidade. Fonte (ARM, 2011)

Então, o problema deixa de existir se o *core* for o único mestre do barramento, pois ele só utiliza o subconjunto AXI4-Lite. A solução proposta foi a arquitetura que se vê na Figura 13. As mudanças propostas são:

- A interface de leitura de instrução do *core* é demultiplexada em um caminho para o SPI *Reader* e outro para a IRAM (Instruction RAM).
    - O endereço de *boot* é da SPI
- O caminho que vai para a IRAM ainda passa por um multiplexador, reaproveitado do PULPino, que arbitra entre a leitura do core e a escrita do SPI *Reader*.
    - A prioridade é para a escrita

- O SPI *Reader* ficará conectado a:
  - Nos pinos externos para a SPI Flash;
  - Na saída do demultiplexador de leitura;
  - No *interconnect* AXI4-Lite.

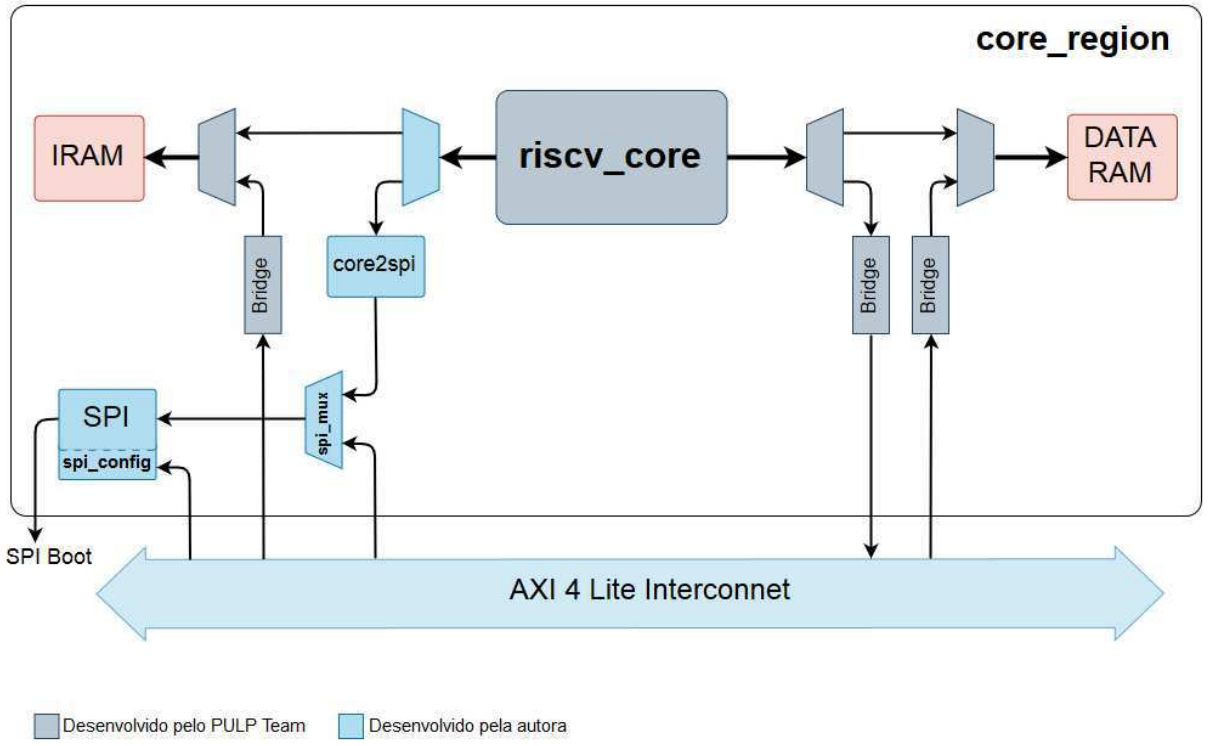

Figura 13: Arquitetura do *core_region*.

Na solução proposta, têm-se uma memória *flash* SPI (*Serial Peripheral Interface*) externa, que é a memória de *boot*, é onde o *core* irá buscar a primeira instrução. Essa memória SPI externa pode, inclusive, substituir a memória de instruções interna ao SoC (IRAM). Porém isso não seria vantagem, já que a memória SPI seria por volta de 40 vezes mais lenta do que a memória interna para que o *core* faça o ciclo de solicitar uma instrução e recebe-lá.

Então, com a arquitetura proposta torna-se possível escrever um *software* que comandará o *core* para, através do barramento AXI4-Lite do *interconnect*, ler uma instrução na memória externa e escrevê-la na memória interna. Este *software*, juntamente com o *software* de aplicação, deverão ser escritos na memória SPI externa da forma que o usuário desejar.

3.2.1 O módulo SPI

De forma genérica, o procolo de comunicação SPI consiste em três sinais: o *clock*, o MOSI (*master out, slave in*) e MISO (*master in, slave out*). A Figura 14 demostra uma transação SPI: síncrono com o *clock*, o mestre envia serialmente um dado e, depois disso, o escravo retorna também serialmente e em sincronia com o *clock* um outro dado. No caso de uma memória, o mestre envia um endereço e o escravo retorna o dado que estava no endereço solicitado.

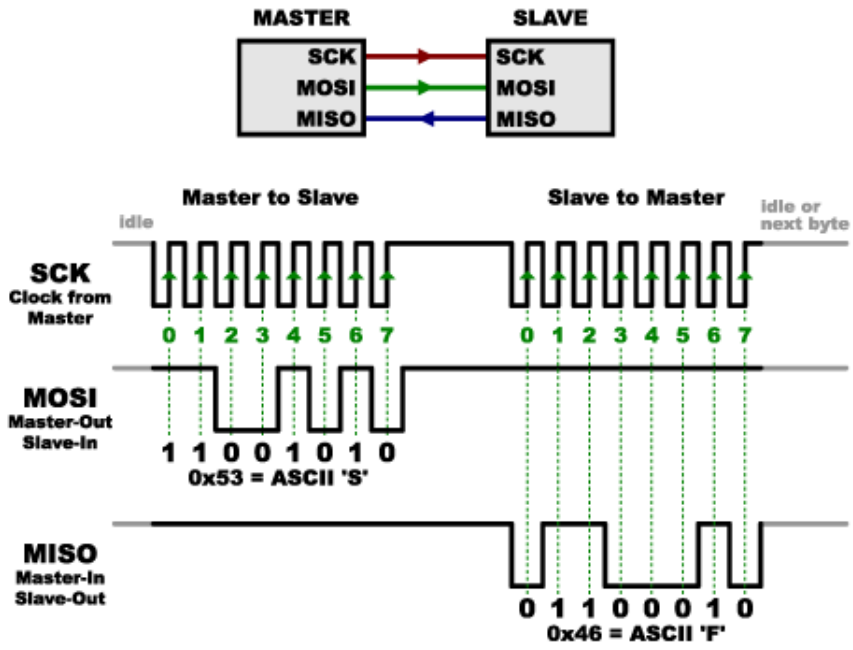

Figura 14: Protocolo SPI.

No caso deste projeto, foi criado uma interface SPI compatível com uma interface AXI4-Lite de leitura. O lado do mestre são comandos que virão do *core*, para solicitar leitura de dados da memória. Para tanto, foi necessário implementar uma tradução do protocolo de comunicação do *core* com a interface AXI4-Lite, tradução esta que será explicitada mais a frente na Seção 3.2.4.

No caso da comunicação com o *chip* de memória SPI, há várias configurações que podem ser feitas, de acordo com o *datasheet* de cada um, por isso, também foi implementado o módulo spi_config (mais detalhes na seção seguinte).

O módulo SPI implementado, então, se comunica com a memória externa SPI, cumprindo seus padrões e responde para o *core* em interface AXI4-Lite.

Na Figura 15, é possível ver a máquina de estado implementada nesse módulo. Basicamente, a transição de estados ou estão cumprindo o protocolo AXI4-Lite ou são transições "diretas" que apenas espera um dado ser enviado serialmente. O funcionamento deste módulo está descrito a seguir.

WAIT_ADDR Estado que espera um *handshake* de endereço. Uma característica que da memória SPI é a possibilidade continuar uma operação de leitura sem a necessidade de enviar novamente a instrução de leitura. Essa configuração pode ser feita através de um sinal interno chamado **m_nibble**, que também depede da operação QUAD SPI. No caso desta máquina de estados, o sinal "*flag_continue*" é o que sinaliza se a memória está configurada para ignorar o envio de instrução ou não. Caso esteja, o próximo estado será SEND_ADDR, caso contrário, o próximo estado será SEND_INSTRUCTION.

SEND_INSTRUCTION Estado para enviar o comando de leitura para a memória SPI, que também pode ser configurável e depende do *datasheet* da memória. Neste caso, as instruções são sempre palavras de 8 bits ou múltiplas de 8. Ao finalizar o envio da instrução, passa-se para o estado SEND_ADDR.

SEND_ADDR Estado para enviar endereço de leitura para a memória. Ao sair deste estado, pode-se ir para um estado de "DUMMY" ou pode-se ir direto para o estado READ_DATA. A transição ou não para o estado "DUMMY" também depende da especificação da memória.

DUMMY Estado necessário para aguardar a leitura.

READ_DATA Estado no qual o escravo retorna para o mestre o dado contido no endereço que havia sido solicitado.

WAIT_RESP Estado que espera o *handshake* entre mestre e escravo, garantindo que o dado foi lido. O próximo estado é aguardar a solicitação de uma leitura em outro endereço, repetindo todo o ciclo novamente, começando do WAIT_ADDR.

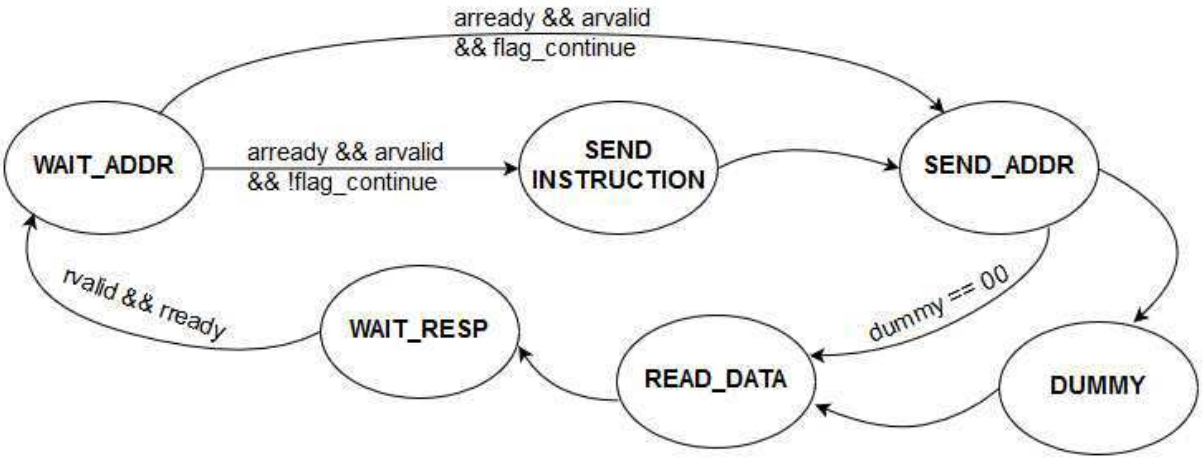

Figura 15: Máquina de estados do módulo SPI.

3.2.2 As configurações SPI

O módulo SPI que se comunica com a memória externa pode ser configurado, via *software*, habilitando configurações intrínsecas ao padrão SPI. O módulo que faz essa configuração é o SPI_CONFIG, que possui a interface AXI4-Lite e que tem o seguinte registrador interno:

Endereço	**Nome**	**Uso**	**Tipo de acesso**
0x00	reg_config	Todas as configurações	Escrita/Leitura

Se o *core* tentar escrever ou ler no spi_config em qualquer endereço que não seja 0x00, o IP retornará uma mensagem de erro no barramento de resposta de escrita ou de leitura.

O registrador reg_config recebe todas as possíveis configurações da comunicação SPI:

31:16	15	14	13:12	11:8	7:0
Não usado	set_config	qe	dummy	m_nibble	instruction

set_config: bit para validar as configurações de quad enable (qe), m_nibble, dummy e instruction

qe: bit para habilitar a leitura QUAD SPI, o valor padrão é 0.

[**3:0**] **m_nibble**: bits para configurar a leitura continua (sem necessidade de repetir o comando de leitura), o valor padrão é 0xA.

[**1:0**] **dummy**: bits para configurar a quantidade de dummy cycles, o valor padrão é 01. Pode ser configurado da seguinte forma:

00 → 0 cycles

01 → 4 cycles

10 → 6 cycles

11 → 8 cycles

[**7:0**] **instruction**: byte para configurar a instrução de leitura. O valor padrão é 0x0B, correspondente a 'FAST READ OPERATION' que faz a comunicação SPI mais simples (porém podendo aceitando a máxima frequência do datasheet das memórias).

3.2.3 O demultiplexador

O demultiplexador desenvolvido seleciona em qual memória o *core* irá buscar uma instrução. Como pode ser visto na Tabela 1, as memórias de instruções estão mapeadas do endereço 0x0000 ao endereço 0xFFFF, sendo necessário apenas 16 *bits*. Isto é, o barramento de endereço de instrução que sai do *core* contém 16 *bits* e seu *bit* mais significativo explicita qual memória ele está querendo acessar. Se o bit mais significativo da memória for 0, então, o *core* lerá uma instrução da IRAM. Se o bit for 1, então, o *core* lerá uma instrução da SPI *Flash*.

A lógica do demultiplexador é muito parecida com a do *interconnect* e aqui também há um protocolo de comunicação. Este protocolo contém os seguintes sinais: *request*, *grant* e *rvalid*. Quando o *core* quer buscar uma instrução ele coloca o sinal *request* em nível alto. Depois, a memória deve responder colocando em nível alto ativo o sinal de *grant*, prometendo para o *core* uma resposta, nesse momento, o *core* pode mudar de endereço mesmo não tendo recebido a instrução referente ao endereço anterior. Quando a memória estiver com a instrução válida, ela envia para o *core* a instrução e o sinal de *rvalid* em nível lógico alto.

Para garantir que o *core* vai receber a resposta da memória correta mesmo já tendo mudado de endereço, há um *buffer* dentro do demultiplexador para guardar o valor do endereço da leitura em andamento.

3.2.4 A integração entre *core* e SPI

Como já foi dito acima, o *core* possui três sinais de controle: *request*, *grant* e *rvalid*, além dos sinais de dado (*rdata*) e endereço (*addr*). Já o módulo SPI foi criado seguindo o protoloco de leitura do AXI4-Lite, o que facilita a integração entre o *core* e o protocolo *SPI*.

Após análise dos sinais de controle do *core*, foi criada a máquina de estados vista na Figura 16. Toda vez que o *core* busca uma instrução, ele coloca o sinal *core_instr_req* em nível lógico alto e passa o endereço da instrução pelo sinal de endereço *core_instr_addr*. Enquanto não há requisição, a máquina de estados fica em WAIT_REQ.

Quando há requisição, é avaliado se o SPI já recebeu o endereço de busca com o sinal *arready*. Se houver o *handshake* de endereço, então o próximo estado será o WAIT_RVALID, se não, será WAIT_ARREADY e nesse estado espera-se o *handshake* acontecer. Quando acontece, o próximo estado também será o WAIT_RVALID.

No estado WAIT_RVALID espera-se o *rvalid*, que é quando a leitura da instrução, de fato, acontece. Após essa leitura, se o *core* estiver solicitando outra instrução, o próximo estado já será WAIT_ARREADY, se não, o próximo estado sera WAIT_REQ.

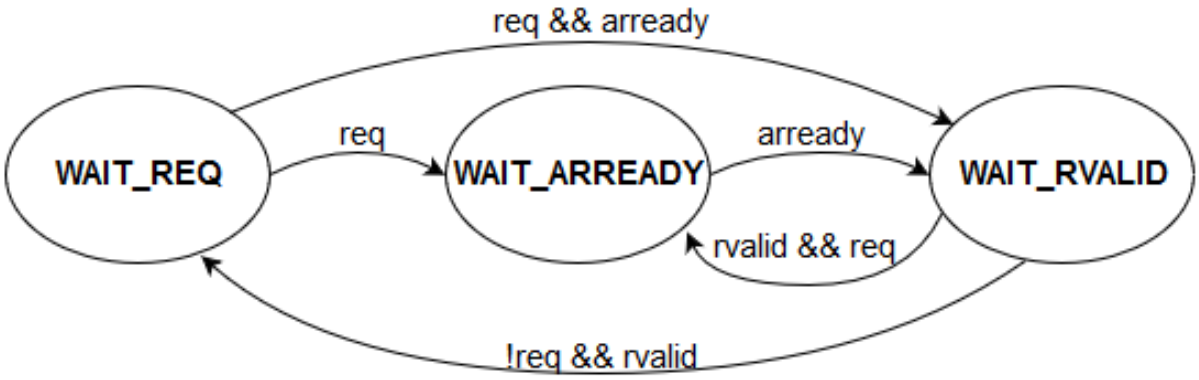

Figura 16: Máquina de estados do CORE2SPI.

3.2.5 O multiplexador para SPI

O módulo spi_mux é um multiplexador que seleciona como o *core* vai ler a memória de externa SPI: através da busca de uma instrução ou através do barramento AXI4-Lite.

Neste caso, a prioridade de leitura sempre será quando o *core* estiver fazendo uma busca de instrução. O outro acesso à memória, via *interconnect*, também é de extrema importância porque é através dele que o *core* conseguirá ler da memória externa para posteriormente escrever na memória interna (IRAM).

Como o único objetivo é fazer com que o *core* tenha acesso de leitura à memória externa para depois copiar o que leu na memória interna, o módulo *spi_mux* sempre retornará erro quando uma operação de escrita for tentada.

4 Resultados

É possível simular o *System on Chip* completo utilizando a ferramenta XCELIUM da *Cadence*.

O projeto PULPino tem um *trace*, que é um módulo para *debug* no qual é possível ver as operações que o *core* executou. Na Figura 17, por exemplo, pode-se encontrar uma parte do *trace* quando o *core* estava executando a rotina de *boot* do SoC.

Destaca-se em vermelho uma iteração do *loop* da rotina de leitura da memória *flash* SPI (operação de *load* (lw) - destacada em verde) e escrita na memória IRAM (operação de *store* (sw) - destacada em verde).

É possível ver que a operação de *load* está ocorrendo no endereço 0x8C88 (apontado pelo registrador PA), que está contido na região de memória alocada para a memória SPI. O valor lido é guardado no registrador interno x10 do *core* e é utilizado na operação de *store*, que acontece no endereço 0x0488, que está contido na região de memória alocada para a IRAM.

```
6756900.00 ns   287815 000081ec 00008067 jalr   x0, x1, 0          x1:0000818c
6758580.00 ns   287899 0000818c fe9414e3 bne    x8, x9, -24        x8:00000322  x9:00000500
6760260.00 ns   287983 00008174 00040513 addi   x10, x8, 0        x10=00000322  x8:00000322
6761100.00 ns   288025 00008178 058000ef jal    x1, 88             x1=0000817c
6762780.00 ns   288109 000081d0 000027b7 lui    x15, 0x2000        x15=00002000
6763620.00 ns   288151 000081d4 00f50533 add    x10, x10, x15      x10=00002322 x10:00000322 x15:00002000
6764460.00 ns   288193 000081d8 00251513 slli   x10, x10, 0x2      x10=00008c88 x10:00002322
6765300.00 ns   288235 000081dc 00052503 lw     x10, 0(x10)        x10=288000ef x10:00008c88  PA:00008c88
6769520.00 ns   288446 000081e0 00008067 jalr   x0, x1, 0          x1:0000817c
6770360.00 ns   288488 0000817c 00050593 addi   x11, x10, 0       x11=288000ef x10:288000ef
6771200.00 ns   288530 00008180 e0040513 addi   x10, x8, -512     x10=00000122  x8:00000322
6772040.00 ns   288572 00008184 00140413 addi   x8, x8, 1          x8=00000323  x8:00000322
6772880.00 ns   288614 00008188 05c000ef jal    x1, 92             x1=0000818c
6774560.00 ns   288698 000081e4 00251513 slli   x10, x10, 0x2     x10=00000488 x10:00000122
6775400.00 ns   288740 000081e8 00b52023 sw     x11, 0(x10)       x11:288000ef x10:00000488  PA:00000488
6776240.00 ns   288782 000081ec 00008067 jalr   x0, x1, 0          x1:0000818c
6777920.00 ns   288866 0000818c fe9414e3 bne    x8, x9, -24        x8:00000323  x9:00000500
6779600.00 ns   288950 00008174 00040513 addi   x10, x8, 0        x10=00000323  x8:00000323
6780440.00 ns   288992 00008178 058000ef jal    x1, 88             x1=0000817c
6782120.00 ns   289076 000081d0 000027b7 lui    x15, 0x2000        x15=00002000
6782960.00 ns   289118 000081d4 00f50533 add    x10, x10, x15      x10=00002323 x10:00000323 x15:00002000
6783800.00 ns   289160 000081d8 00251513 slli   x10, x10, 0x2      x10=00008c8c x10:00002323
6784640.00 ns   289202 000081dc 00052503 lw     x10, 0(x10)        x10=eee90693 x10:00008c8c  PA:00008c8c
6788860.00 ns   289413 000081e0 00008067 jalr   x0, x1, 0          x1:0000817c
6789700.00 ns   289455 0000817c 00050593 addi   x11, x10, 0       x11=eee90693 x10:eee90693
6790540.00 ns   289497 00008180 e0040513 addi   x10, x8, -512     x10=00000123  x8:00000323
6791380.00 ns   289539 00008184 00140413 addi   x8, x8, 1          x8=00000324  x8:00000323
6792220.00 ns   289581 00008188 05c000ef jal    x1, 92             x1=0000818c
6793900.00 ns   289665 000081e4 00251513 slli   x10, x10, 0x2     x10=0000048c x10:00000123
6794740.00 ns   289707 000081e8 00b52023 sw     x11, 0(x10)       x11:eee90693 x10:0000048c  PA:0000048c
6795580.00 ns   289749 000081ec 00008067 jalr   x0, x1, 0          x1:0000818c
6797260.00 ns   289833 0000818c fe9414e3 bne    x8, x9, -24        x8:00000324  x9:00000500
6798940.00 ns   289917 00008174 00040513 addi   x10, x8, 0        x10=00000324  x8:00000324
6799780.00 ns   289959 00008178 058000ef jal    x1, 88             x1=0000817c
6801460.00 ns   290043 000081d0 000027b7 lui    x15, 0x2000        x15=00002000
6802300.00 ns   290085 000081d4 00f50533 add    x10, x10, x15      x10=00002324 x10:00000324 x15:00002000
```

Figura 17: Processo de cópia do do conteúdo damemória SPI para a IRAM.

Depois de copiar toda a memória de instruções, o *core* começa a executar as instruções que ele copiou. Na Figura 18, pode-se ver parte da rotina do AES sendo executada. É importante notar que o "PA" (*Point Address*) está apontando para a posição

do IP AES no mapa de memórias (Tabela 1).

```
16008380.00 ns  750389 00000460 00912223 sw    x9, 4(x2)          x9:00000000  x2:00107ff0  PA:00107ff4
16008400.00 ns  750390 00000464 ccccd437 lui   x8, 0xccccd000     x8=ccccd000
16008420.00 ns  750391 00000468 aaaab4b7 lui   x9, 0xaaaab000     x9=aaaab000
16008440.00 ns  750392 0000046c 01212023 sw    x18, 0(x2)         x18:00000000  x2:00107ff0  PA:00107ff0
16008460.00 ns  750393 00000470 d0c68693 addi  x13, x13, -756     x13=0f0e0d0c x13:0f0e1000
16008480.00 ns  750394 00000474 eeeef937 lui   x18, 0xeeeef000    x18=eeeef000
16008500.00 ns  750395 00000478 90860613 addi  x12, x12, -1784    x12=0b0a0908 x12:0b0a1000
16008520.00 ns  750396 0000047c 50458593 addi  x11, x11, 1284     x11=07060504 x11:07060000
16008540.00 ns  750397 00000480 10050513 addi  x10, x10, 256      x10=03020100 x10:03020000
16008560.00 ns  750398 00000484 00112623 sw    x1, 12(x2)          x1:00000140  x2:00107ff0  PA:00107ffc
16008580.00 ns  750399 00000488 288000ef jal   x1, 648            x1=0000048c
16008620.00 ns  750401 00000710 191027b7 lui   x15, 0x19102000    x15=19102000
16008640.00 ns  750402 00000714 00078713 addi  x14, x15, 0        x14=19102000 x15:19102000
16008660.00 ns  750403 00000718 00a7222b p.sw  x10, 4(x14!)       x14=19102004 x10:03020100 x14:19102000
PA:19102000
16008680.00 ns  750404 0000071c 00878513 addi  x10, x15, 8        x10=19102008 x15:19102000
16008740.00 ns  750407 00000720 00c78793 addi  x15, x15, 12       x15=1910200c x15:19102000
16008760.00 ns  750408 00000724 00b72023 sw    x11, 0(x14)        x11:07060504 x14:19102004  PA:19102004
16008780.00 ns  750409 00000728 00c52023 sw    x12, 0(x10)        x12:0b0a0908 x10:19102008  PA:19102008
16008860.00 ns  750413 0000072c 00d7a023 sw    x13, 0(x15)        x13:0f0e0d0c x15:1910200c  PA:1910200c
16008940.00 ns  750417 00000730 00008067 jalr  x0, x1, 0          x1:0000048c
16009000.00 ns  750420 00000048 eee90693 addi  x13, x18, -274     x13=eeeeeeee x18:eeeef000
16009020.00 ns  750421 00000490 aaa48593 addi  x11, x9, -1366     x11=aaaaaaaa  x9:aaaab000
16009040.00 ns  750422 00000494 fff00613 addi  x12, x0, -1        x12=ffffffff
16009060.00 ns  750423 00000498 ccc40513 addi  x10, x8, -820      x10=cccccccc  x8:ccccd000
16009080.00 ns  750424 0000049c 2c0000ef jal   x1, 704            x1=000004a0
16009120.00 ns  750426 0000075c 191027b7 lui   x15, 0x19102000    x15=19102000
16009140.00 ns  750427 00000760 02078893 addi  x17, x15, 32       x17=19102020 x15:19102000
16009160.00 ns  750428 00000764 02478813 addi  x16, x15, 36       x16=19102024 x15:19102000
16009180.00 ns  750429 00000768 02878713 addi  x14, x15, 40       x14=19102028 x15:19102000
16009200.00 ns  750430 0000076c 02c78793 addi  x15, x15, 44       x15=1910202c x15:19102000
16009220.00 ns  750431 00000770 00a8a023 sw    x10, 0(x17)        x10:cccccccc x17:19102020  PA:19102020
16009240.00 ns  750432 00000774 00b82023 sw    x11, 0(x16)        x11:aaaaaaaa x16:19102024  PA:19102024
16009320.00 ns  750436 00000778 00c72023 sw    x12, 0(x14)        x12:ffffffff x14:19102028  PA:19102028
16009400.00 ns  750440 0000077c 00d7a023 sw    x13, 0(x15)        x13:eeeeeeee x15:1910202c  PA:1910202c
```

Figura 18: O AES em funcionamento.

Por fim, é interessante ressaltar também que o SoC foi prototipado em FPGA, através do *softwares Synplify* e *Quartus*. Para testá-lo, foi utilizado o *software* Energia para programar um microcontrolador que por sua vez escreveu os programas (*boot* e aplicações) na memória externa. Assim, foi possível testar algumas funcionalidades do SoC e validar seu funcionamento.

Considerações finais

Com este trabalho foi possível aprender mais sobre protolocos de comunicação, como SPI e AMBA AXI, como também sobre a interface entre *hardware* e *software* de um *System on Chip*. Também foi uma oportunidade de consolidar o conhecimento em *design* de *hardware* fazendo os módulos para a integração do SoC.

Com simulação e prototipação do SoC funcionando corretamente, é possível concluir que a integração foi realizada com sucesso.

Este projeto também permitiu que todos seus colaboradores pudessem ser capacitados em realizar projetos de microeletrônica. Foi um projeto grande e complexo o suficiente para ser divido em equipes e para que as equipes pudessem crescer em conjunto.

Além disso, o projeto propiciou o treinamento em ferramentas computacionais bastante utilizadas no âmbito industrial, como as ferramentas fornecidas pela *Cadence* e pela *Synopsys*.

É importantíssimo constatar que este nível de projeto nos mostra que somos capazes de produzir tecnologia de alto nível, passível de ser reconhecido internacionalmente. Afinal, neste trabalho, utilizamos um *core* RISC-V, que vem ganhando cada vez mais força no mercado, chegando a assombar empresas consolidadas no mercado como a ARM.

Um projeto de tamanha complexidade nos coloca em sintonia com o que as melhores universidades do mundo estão desenvolvendo. Trabalhamos com uma tecnologia desenvolvida pela Universidade da Califórnia, em Berkeley, que é uma das maiores referências no assunto. Essa tecnologia também foi explorada pela ETH Zurich no projeto PULPino, o qual foi utilizado como base para o desenvolvimento deste projeto, que apesar de ser um grande desafio, nos mostrou que equiparáveis aos grande centros de tecnologia do mundo.

Referências

ARM. *AMBA AXI and ACE Protocol Specification*. USA, 2011. Citado 2 vezes nas páginas 26 e 37.

HOROWITZ, P.; HILL, W. *The Art of Electronics*. 3. ed. 32 Avenue of the Americas, New York, NY 10013-2473, USA: Cambridge University Press, 2016. Citado na página 24.

STALLINGS, W. *COMPUTER ORGANIZATION AND ARCHITECTURE DESIGNING FOR PERFORMANCE*. 9. ed. New Jersey, USA: PEARSON, 2013. Citado 2 vezes nas páginas 20 e 21.

TRABER, A.; GAUTSCHI, M. *PULPino: Datasheet*. ETH Zurich and University of Bologna, 2017. Citado na página 27.

WESTE, N. H. E.; HARRIS, D. M. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4. ed. [S.l.]: PEARSON, 2011. Citado 2 vezes nas páginas 19 e 20.